On-line Distributed Debugging on Scaleable Multicomputer Architectures*

Thomas Bemmerl[1] and Roland Wismüller[2]

[1] Intel Corp., European Supercomputer Dev. Center, D-85622 Feldkirchen, FRG
RWTH Aachen, Lehrstuhl für Betriebssysteme (LfBS), D-52056 Aachen, FRG
[2] Technische Universität München, Institut für Informatik, D-80290 München, FRG

Abstract. Debugging parallel programs is one of the most tedious jobs in programming scalable multiprocessor architectures. Due to the distributed resources of these machines, programming is often architecture dependent. Most development tools still reflect this dependency even during the analysis phase of parallel programs. This paper presents the distributed debugger DETOP, which offers a global name space and hides architectural features like the mapping of processes. DETOP is part of the integrated tool environment TOPSYS implemented on iPSC hypercubes, networks of SPARCstations and partly on Transputer systems.

1 Introduction

Some compute intensive applications, e.g. fluid dynamics and simulations, require computing performance far beyond the performance of today's supercomputers. Many computer scientists claim that the performance necessary for these "grand challenges" can only be reached with parallel distributed memory computers.

Although some of these computers already offer hardware concepts to hide the specific architectural features (e.g. topology is hidden by cut-through routing and virtual channels), programming these machines is still more time consuming than writing programs for workstations [5]. A lack in adequate programming tools for parallel architectures boosts this effect. In particular the design and implementation of on-line debuggers is a challenging task due to the following problems:

- The distribution of resources, time relationship between resources and consistency of states.
- The complexity of the target system (large number of processes, processors, channels).
- Highly dynamic process models and applications.
- The natural non-determinism of parallel programs.
- The interference between debugger and user program, that may change its dynamic behaviour (probe effect).

* Partly funded by the German Science Foundation, Contract: SFB 0342

2 State of the Art and Related Work

The techniques to solve these problems can be divided into four main categories explained below.

Off-line Debugging. An off-line debugging session consists of two phases. In the execution phase information on the program's behaviour is recorded in trace files for each node or process. In the second phase, these traces can be analyzed with tools that offer either a graphical data representation or database like information retrieval. Since all information is acquired within a single program run, nondeterminism is not a problem. However, storing detailed traces needs an enormous amount of memory (and time).

On-line Debugging. On-line (interactive) debuggers allow to stop the target program at any point of the user's interest (breakpoint). Once the program is halted, its state can be examined or modified. Finally the program is continued until another breakpoint occurs. To get information on previous states, the program is usually restarted ("cyclic debugging"), so techniques for deterministic re-execution are useful.

Several methods to specify distributed event predicates triggering a breakpoint have been proposed, e.g. behavioural abstraction [1] or event based approaches [8, 6]. Once the breakpoint is triggered, all processes must be stopped in a consistent state, i.e. a state that may occur during normal execution. Stopping should be as fast as possible, since if a large number of statements is executed after the event occurrence, the program's state may no longer be useful.

Deterministic Re-execution. Since the order of messages arriving at a process may depend on occasional delays, parallel programs are usually nondeterministic, i.e. may behave differently when re-executed. "Instant replay" [7] achieves deterministic re-execution by logging the order of messages in the first program run. During re-execution, the original message order is enforced by delaying the receipt of early messages.

Static Analysis. In contrast to above techniques, static analysis allows to detect some classes of program errors (e.g. deadlocks or access anomalies) without ever executing the program. Since static analysis examines all possible executions, even errors are detected that don't occur in a particular program run.

In the following chapter, we will focus on the event based approach and introduce a transformation-based method for efficient event recognition.

3 Event Based Debugging with the DETOP Debugger

3.1 Programming and Debugging Model

We assume a programming model based on light weight processes and message passing as provided by the programming library MMK [4]. MMK offers three kinds of programming objects, that can be created and deleted dynamically:

Tasks, i.e. light weight processes for computing, *mailboxes* for communication and *semaphores* for synchronization. Objects can be accessed independently of their current location in the distributed system by means of globally unique identifiers.

This location transparency simplifies both program coding and debugging, since the user need not have in mind the location of the programming objects. However, the debugger must provide a global view of the distributed program and some kind of (virtual) global time. The ability to dynamically create and delete objects also influences the debugger. The problems to solve are how to name these objects and how to keep the debugger informed of all existing objects and their locations without a large communication overhead.

To support parallel debugging, DETOP adopts the event action model. The user can specify events, e.g. receipt of a message or execution of a statement, and relationships between events. DETOP uses two different methods to define these relationships: On the user level, a simple event description language is used, while event recognition is based on deterministic finite automata. Once events, relationships and actions have been defined, the debugger permanently monitors the specified events. When they satisfy the relationship, the actions are performed, e.g the program is halted. This scheme helps users to move from sequential to parallel programming environments, since it is a natural extension of sequential debugging methods. It may also be used for other purposes, e.g. performance monitoring or visualization.

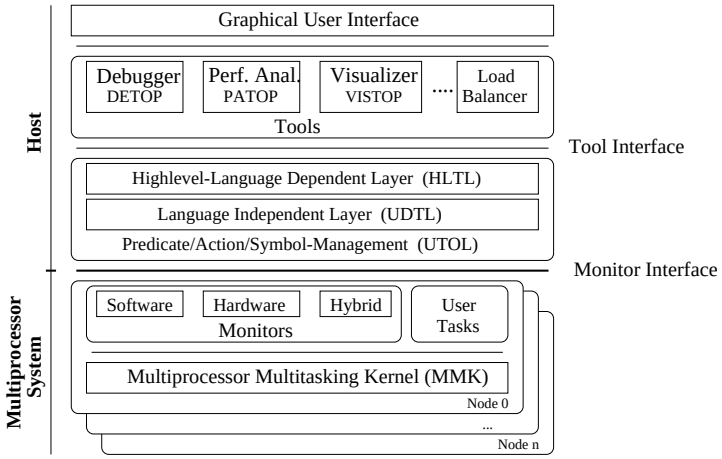

Fig. 1. Hierarchical model of TOPSYS

3.2 Architecture of the DETOP Debugging System

DETOP is part of the TOPSYS programming environment [2], that is built in a hierarchical manner, as shown in fig. 1. We have defined two important interfaces:

The **monitor interface** allows to observe a target program at the machine level. TOPSYS uses a distributed monitoring system with a monitor attached to

each node of the multicomputer. Each monitor recognizes local events (occurring on the monitor's node), tests relationships between these events and performs local actions. To implement global predicates and actions, the monitors can exchange messages via a virtually fully connected communication network. Beside a pure software implementation, there is also a hybrid version that uses special hardware and a dedicated communication bus to detect and combine events [3].

The **tool interface** offers the same functionality at the level of source language and programming model and provides a global view of the distributed system. So while the tool interface allows to specify global events related to the source code, e.g. 'execution of line 23 in file foo.c' or 'sending a message to mailbox A', the monitoring system is based on machine code and hooks in the operating system, e.g. 'execution of address $1e34' or 'insertion into send queue'.

A central layer named UTOL is responsible for the transformations between these abstraction levels and the management of predicates, actions and symbol tables. To increase portability there are two sublayers: the high-level language dependent part (HLTL) contains parsers for commands and expressions and also handles complex data types. The language independent layer (UDTL) translates symbolic names into addresses and vice-versa. It also implements the predicate transformation described in the next chapter.

Execution Events	Data Events	Communication Events
execute(task,procedure)	write(task,variable)	send(task,mailbox,msg_ptr)
execute(task,line)	read(task,variable)	receive(task,mailbox)
	assign(task,variable,value)	request(task,semaphore,units)
		release(task,semaphore,units)

Table 1. User level events (wildcards for objects are allowed)

3.3 Monitoring of Global Predicates

As stated before, we use a simple event description language to define global predicates at the user level. A predicate consists of an arbitrary number of events (see table 1) that may be combined with any of the following operators:

$\vee$: $A \vee B$ becomes TRUE when A has become TRUE or B has become TRUE
$\wedge$: $A \wedge B$ becomes TRUE when both A and B have become TRUE
$\rightarrow$: $A \rightarrow B$ becomes TRUE when B becomes TRUE after A has become TRUE

In contrast to other approaches our monitoring system does not directly interpret the global user predicate. Instead, each monitor processes only those parts of the predicate that can be supervised locally. These parts are determined by inspecting the objects that participate in an event.

The mapping between object identifiers and node numbers is established by a localizer module within the UTOL. Since objects may be created and deleted dynamically, an object's node cannot be determined without communication with the target system. In order to keep this communication infrequent, we use a cache-based approach. The localizer holds an internal table containing object identifiers and corresponding node numbers. If an identifier is found in

this table, no communication is needed. In the other case, the localizer requests each monitor to send the identifiers of all existing objects on its node and updates the table. If an object has been deleted, but is still in the localizer table, a monitor may receive an event description referring a non-existent object. In this case the event description can be ignored, since the event can never occur.

To achieve distributed global event detection we use a concept called "crosstriggers". Crosstriggers are flags that are initially FALSE and can only be set to TRUE. They are used to implement both communication between monitors and the '$\vee$' operator. For example, assume that $P = (a_0 \rightarrow b_1) \vee (c_2 \rightarrow d_2)$, where the indices indicate the nodes that can detect the subpredicates. Distributing this predicate using crosstriggers results in $P = CT0$; $CT1 \mid= a_0$; $CT0 \mid= CT1_1 \rightarrow b_1$; $CT0 \mid= c_2 \rightarrow d_2$. The '$\mid=$' operator denotes that the crosstrigger on the left hand side is set when the right hand side evaluates to TRUE.

The software monitor implements crosstriggers as messages, where $CTx \mid= P$ means "associate action 'send message of type x' with predicate P", while CTx_i on a right hand side denotes 'message of type x arrives at node i'. In the hardware monitor a crosstrigger is a single wired-or signal in a special bus.

Although the set of local predicates could be monitored directly, we decided to compile it into a deterministic Mealy type automaton. This automaton, that generates sequences of local actions as its output, is built by simulating the parallel evaluation of all predicates on a node. This step requires additional computing effort when a predicate is defined. However, it reduces the monitoring overhead, since the event detection algorithm is now a simple automaton interpreter, that can also be efficiently implemented in hardware. In addition, predicate transformation may include some optimization, e.g. to reduce communication, so the scalability of the monitoring scheme is further increased.

4 The Distributed Debugger DETOP

Based on the monitoring scheme introduced above, we have built the distributed debugger DETOP. The debugger uses a graphical interface offering commands that fall into five major groups:

1. **Execution control:** This group contains commands to start, stop and single step tasks.
2. **Inspection of program state:** These commands allow to retrieve information about the program's state, e.g. a list of all currently existing objects, their mapping to processing nodes, scheduling state of tasks, message and task queues of mailboxes, value and type of variables.
3. **Modification of program state:** Currently only modification of variables is supported.
4. **Breakpoints and traces:** As stated in chapter 3.3 distributed breakpoints can be defined based on the events in table 1. The user can also specify events that will be logged to a trace. Starting and stopping the trace can be controlled by a predicate with an associated trace trigger action.

5. **Miscellaneous:** Beside configuration menus and on-line help, multiple listing windows can be used to view source code, to define execution events and to display the current line of execution of a task.

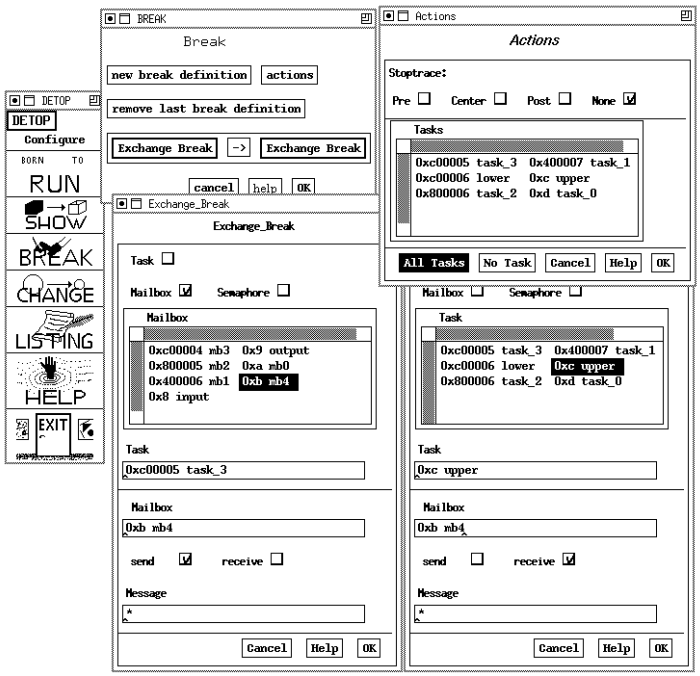

Fig. 2. Definition of a breakpoint with DETOP

Figure. 2 shows how breakpoint definitions are supported by the user interface. In the small window in the upper left, the user defines the overall structure of the predicate, i.e. chooses between different event types and operators. The detailed definition of each event and the associated actions is done in separate windows in order to keep the display clear. All objects are accessed with symbolic names whenever possible. In the example, all tasks are stopped, when task *upper* receives a message from mailbox *mb4* that has been sent by *task_3*.

References

1. P. Bates and J. C. Wileden, "An Approach to High-Level Debugging of Distributed Systems", In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, vol. 18(8) of *ACM SIGPLAN Notices*, pp. 107–111, Aug. 1983.
2. T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, and R. Wismüller, "The Design and Implementation of TOPSYS", Report 342/16/91 A, Technische Universität München, July 1991.

3. T. Bemmerl, R. Lindhof, and T. Treml, "The Distributed Monitoring System of TOPSYS", In *Proc. CONPAR 90 - VAPP IV Conference*, Zürich, Schweiz, Sept. 1990.
4. T. Bemmerl and T. Ludwig, "MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing", In *Proc. CONPAR 90 - Vapp IV Conference*, Zürich, Schweiz, Sept. 1990.
5. G. C. Fox, M. Johnson, G. Lyzenga, O. S. W., J. Salmon, and D. Walker, *Solving problems on concurrent processors*, Prentice Hall, Englewood Cliffs, 1988.
6. D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed Systems", In *Proc. Twenty-First Annual Hawaii International Conference on System Sciences*, vol. II, Sortware Track, pp. 166–175, 1988.
7. T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Trans. Comput.*, vol. C-35, no. 4, pp. 471–481, Apr. 1987.
8. B. P. Miller and J. D. Choi, "Breakpoints and Halting in Distributed Programs", In *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 316–323, Los Alamitos, California, 1988.

This article was processed using the LaTeX macro package with LLNCS style